

Języki i Paradygmaty Programowania

Programowanie imperatywne w Haskellu

Marcin Benke

12 marca 2018

Programowanie imperatywne

Istotę programowania imperatywnego stanowią przede wszystkim:

- obliczenia z różnego rodzaju efektami ubocznymi
 - globalny stan i jego modyfikacja
 - szeroko rozumiane I/O
 - wyjątki
 - ...
- sekwencjonowanie obliczeń: zrób to, a potem zrób tamto.

```
do { print A ; print B }
```

Obliczenia funkcyjne

Zasada przejrzystości zapewnia, że obliczenie wyrażenia da zawsze ten sam wynik, możemy zatem fragmenty programu wykonywać w dowolnej kolejności, a nawet równolegle.

Jak w takim modelu mieszczą się obliczenia imperatywne, sekwencyjne?

```
do { print A ; print B }
```

Obliczenia funkcyjne

Możemy zdefiniować typ, którego wartościami będą obliczenia, z operacjami:

- Obliczenie czyste (daje jakąś wartość, nie ma efektów ubocznych)
- Sekwencjonowanie obliczeń: `obliczenie -> (wynik -> obliczenie) -> obliczenie`
- Operacje pierwotne (np. wczytaj, wypisz, etc.)

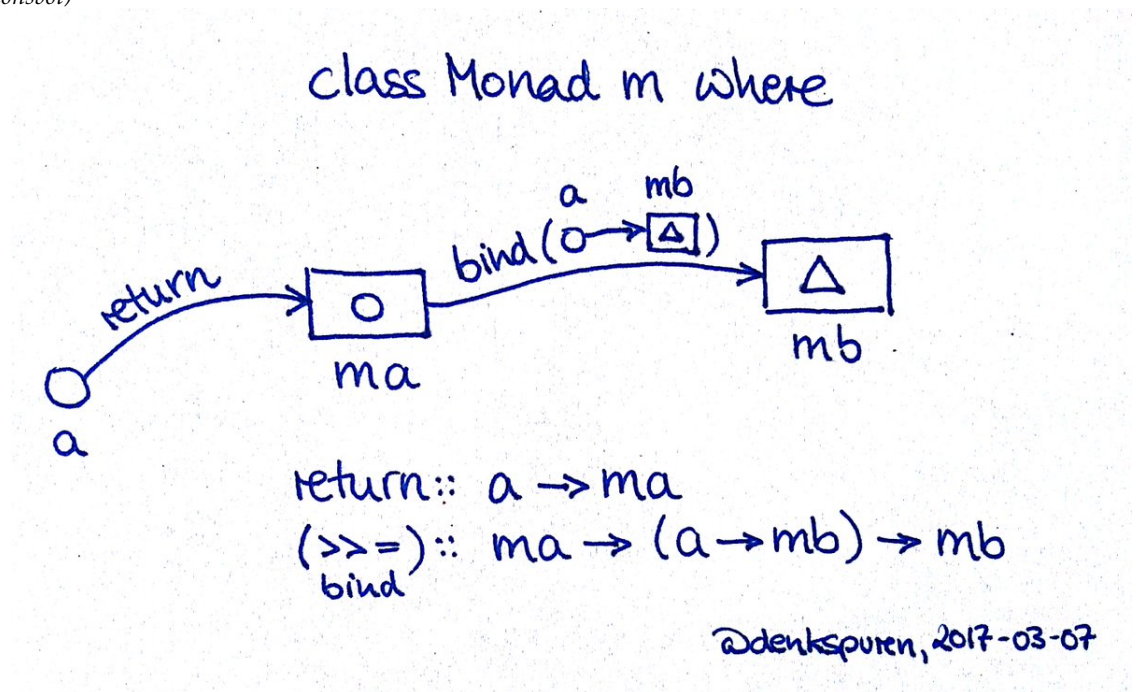
Mechanizm wykonywania obliczeń ("interpreter", maszyna wirtualna)

Klasa Monad czyli programowalny średnik

```
class Monad obliczenie where
  return :: a → obliczenie a
  (>>=) :: obliczenie a → (a → obliczenie b)
           → obliczenie b
```

- Klasa **Monad** jest klasą konstruktorową (jak **Functor**).
- Gdy **m** jest instancja **Monad**, to **m a** jest typem obliczeń o wyniku typu **a**.
- **return x** jest czystym obliczeniem dającym wynik **x**
- Operator **(>>=)** (zwany "bind") sekwencjonuje obliczenia ("programowalny średnik")

Monads: just a fancy name for scripting your semicolons (via @TacticalGrace, inspired by @donsbot)



Skróty notacyjne

Jeżeli kolejne obliczenie nie korzysta z wyniku (a tylko z efektu) poprzedniego, możemy użyć operatora **(>>)**

```
o1 >> o2 = o1 >>= \_ -> o2
```

```
print A >> print B
```

Ponadto czasami wygodniej jest zapisywać złożenie obliczeń w kolejności analogicznej do złożenia funkcji:

```
f =<< o = o >>= f
```

Najprostszy efekt: brak efektu

Najprostsza monada jest **Identity** (moduł **Control.Monad.Identity** w bibliotece standardowej)

```
newtype Identity a = Identity { runIdentity :: a }
```

```
instance Monad Identity where
  return a          = Identity a    -- return = id
  (Identity x) >>= f = f x           -- x >>= f = f x
```

Trzy prawa monadyki

Każda monada musi spełniać następujące prawa:

1. $(\text{return } x) \gg= f == f \ x$
 2. $m \gg= \text{return} == m$
 3. $(f \gg= g) \gg= h == f \gg= (g \gg= h)$
- gdzie $f \gg= g = (\lambda x \rightarrow (f \ x \gg= g))$

Pierwsze dwa prawa mówią, że *return* nie ma efektów.

Trzecie prawo mówi, że sekwencjonowanie obliczeń jest łączne, czyli w pewnym sensie, że

```
(o1;o2);o3 == o1;(o2;o3)
```

...i możemy je traktować jako sekwencję $o1;o2;o3$

Podobnie jak zapis $a + b + c$ jest jednoznaczny dzięki łączności dodawania.

Prosty efekt: obliczenia zawodne

Cel: chcemy modelować obliczenia, które czasem czasem zawodzą

Środek: monada **Maybe**

```
instance Monad Maybe where
  return x = Just x
  Nothing >>= k = Nothing
  Just x   >>= k = k x
```

- Obliczenie, które nie daje wyniku: **Nothing**
- Obliczenie, które daje wynik x: **Just x**
- Jeśli pierwsze obliczenie zawodzi, to cała sekwencja zawodzi

Korzyści z monady **Maybe**

Możemy oczywiście korzystać z **Maybe** bez mechanizmu monad:

```
case obliczenie1 of
  Nothing → Nothing
  Just x  → case obliczenie2 of
              Nothing → Nothing
              Just y  → obliczenie3
```

Monada pozwala nam to zapisać zgrabniej:

```
obliczenie1 >>= (λx →
  obliczenie2 >>= (λy →
    obliczenie3))
```

A nawet jak się przekonamy za chwilę, jeszcze zgrabniej...

Notacja **do**

W obliczeniach monadycznych często pojawia się kod typu

```
obliczenie1 >>= (λx →
  obliczenie2 >>= λy →
    obliczenie3))
```

Dla usprawnienia zapisu oraz dla podkreślenia potencjalnej imperatywności możemy zapisywać je przy pomocy notacji **do**:

```
do {
  x ← obliczenie1;
  y ← obliczenie2;
  obliczenie3
}
```

Uwaga

do jest tylko równoważną notacją, nie powoduje wykonania efektu.

Przykłady

```
$ ghci
> import Text.Read

> do { n <- readMaybe "ala" ; return (n+1) }
```

Nothing

```
> do {n <- readMaybe "41" ; return (n+1) }  
Just 42
```

```
> readMaybe "41" >=> return . (+1)  
Just 42
```

Layout i do

do możemy zapisywać też przy pomocy wcięć, czyli zamiast

```
do {  
  x ← obliczenie1;  
  y ← obliczenie2;  
  obliczenie3  
}
```

możemy napisać

```
do  
  x ← obliczenie1  
  y ← obliczenie2  
  obliczenie3
```

do i let

Konstrukcja

```
do { fragment1; let {x=e}; fragment2 }
```

jest równoważna

```
do { fragment1; let x = e in do { fragment2 } }
```

Przykład:

```
main = do  
  input ← getLine  
  let output = map toUpper input  
  putStrLn output
```

...oczywiście moglibyśmy to zapisać krócej:

```
main = getLine >>= putStrLn o map toUpper
```

do i dopasowanie wzorca

Podobnie jak możemy napisać `let (x:xs) = foo in ...` możemy w bloku **do** napisać `(x:xs) <- foo`

```
do
  (x:xs) <- foo
  ys <- bar xs
  return (x:ys)
```

Co się stanie jeśli wynik `foo` jest listą pustą?

Otóż klasa **Monad** definiuje jeszcze jedną metodę:

```
fail :: (Monad m) => String -> m a
```

...w przypadku gdy dopasowanie wzorca zawiedzie jest ona wywoływana ze stosownym komunikatem.

Zachowanie metody **fail** zależy oczywiście od konkretnej monady.

Obsługa błędów

Przeważnie w wypadku błędu chcemy mieć więcej informacji niż tylko, że obliczenie zawiodło: komunikat o błędzie.

Możemy do tego wykorzystać typ **Either**:

```
instance Monad (Either error) where
  return = Right
  (Left e)  >>= _ = Left e
  (Right x) >>= k = k x
```

```
> Left "error" >> return 3
Left "error"
```

```
> do {n <- readEither "41" ; return (n+1) }
Right 42
```

Obsługa błędów: MonadError

Możemy też abstrakcyjnie zdefiniować protokół obsługi błędów:

```
class (Monad m) => MonadError e m | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

```
instance (Error e) => MonadError e (Either e) ...
```

```
class Error a where
  noMsg :: a
  strMsg :: String -> a
```

Wskazówka

Klasy `Error` i `MonadError` są zdefiniowane w module `Control.Monad.Error`

MonadError — przykład

```
data ParseError = Err {location::Int,
                      reason::String}
instance Error ParseError ...
type ParseMonad = Either ParseError
parseHexDigit :: Char → Int → ParseMonad Integer
parseHex :: String → ParseMonad Integer
toString :: Integer → ParseMonad String

-- convert zamienia napis z liczba szesnastkowa
--      na napis z liczba dziesiętna
convert :: String → String
convert s = str where
  (Right str) = tryParse s `catchError` printError
  tryParse s = do {n ← parseHex s; toString n}
  printError (Err loc msg) = return $ concat
    ["At index ", show loc, ":", msg]
```

Nowszy wariant: Except

Począwszy od GHC 7.8 mamy `Control.Monad.Except`

```
type Except e = ...
runExcept :: Except e a → Either e a
instance MonadError e (Except e)
```

(w pewnym przybliżeniu, rzeczywistość jest nieco bardziej skomplikowana)

Monada IO

- Typ `IO` jest wbudowanym typem reprezentującym obliczenia, które mogą mieć (szeroko rozumiane) efekty wejścia-wyjścia.
- Na przykład `getLine::IO String` jest obliczeniem, które pobiera linię ze standardowego wejścia, a jego wynikiem jest pobrana linia.
- Funkcja `main` powinna być typu `IO ()`.
- Wykonanie funkcji `main` powoduje realizację związanych z nią efektów, np.

```
main = print 42
```

spowoduje wypisanie na stdout (reprezentacji tekstowej) liczby 42.

Ważniejsze funkcje IO

```
print      :: Show a => a -> IO ()
putStrLn  :: String -> IO ()
putChar   :: Char -> IO ()
putStr    :: String -> IO ()
getChar   :: IO Char
getline   :: IO String
getContents :: IO String
getArgs  :: IO [String] -- System.Environment
```

Funkcja **getContents** daje całą zawartość wejścia jako leniwą listę. Funkcja **interact** może być zdefiniowana właśnie przy użyciu **getContents**:

```
interact :: (String -> String) -> IO ()
interact f = do s <- getContents
              putStr (f s)
```

Operacje na plikach i deskryptorach

Moduł System.IO

```
type FilePath = String
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

```
data Handle = ...
stdin, stdout, stderr :: Handle
hClose :: Handle -> IO ()
hFlush :: Handle -> IO ()
```

```
hGetChar :: Handle -> IO Char
hGetContents :: Handle -> IO String
hPutStr :: Handle -> String -> IO ()
```

Operacje na plikach i deskryptorach

```
withFile :: FilePath -> IOMode ->
           (Handle -> IO r) -> IO r
openFile :: FilePath -> IOMode -> IO Handle
```

```
data IOMode = ReadMode
            | WriteMode
            | AppendMode
            | ReadWriteMode
```



```

hSetBuffering :: Handle → BufferMode → IO ()
data BufferMode = NoBuffering
                | LineBuffering
                | BlockBuffering (Maybe Int)

```

Więcej — patrz dokumentacja modułu **System.IO**

Użyteczne kombinatory monadyczne

...w większości zdefiniowane w module **Control.Monad**

```

mapM :: Monad m ⇒ (a → m b) → [a] → m [b]
mapM_ :: Monad m ⇒ (a → m b) → [a] → m ()
forM :: Monad m ⇒ [a] → (a → m b) → m [b]
forM_ :: Monad m ⇒ [a] → (a → m b) → m ()
sequence :: Monad m ⇒ [m a] → m [a]
sequence_ :: Monad m ⇒ [m a] → m ()
liftM :: Monad m ⇒ (a1 → r) → m a1 → m r -- fmap
liftM2 :: Monad m ⇒ (a1 → a2 → r)
        → m a1 → m a2 → m r
...

```

Na przykład

```

void(forM [1..7] print)
forM_ ['1'..'7'] putChar >> putStrLn ""
liftM2 (+) (readMaybe "40") (readMaybe "2")

```

Monad rozszerza Applicative (od niedawna)

```

class Applicative m ⇒ Monad m where ...

```

Zauważmy że

```

ap :: Monad m ⇒ m (a → b) → m a → m b
ap mf ma = do { f ← mf; a ← ma; return (f a) }

```

stąd w nowszych wersjach GHC możemy zobaczyć komunikat

Warning:

```

'Parser' is an instance of Monad but not Applicative
- this will become an error in GHC 7.10,
  under the Applicative-Monad Proposal.

```